Package 'ambient'

March 21, 2020

Type Package

Title A Generator of Multidimensional Noise

Version 1.0.0

Maintainer Thomas Lin Pedersen <thomasp85@gmail.com>

Description Generation of natural looking noise has many application within simulation, procedural generation, and art, to name a few. The 'ambient' package provides an interface to the 'FastNoise' C++ library and allows for efficient generation of perlin, simplex, worley, cubic, value, and white noise with optional pertubation in either 2, 3, or 4 (in case of simplex and white noise) dimensions.

License MIT + file LICENSE

Encoding UTF-8

SystemRequirements C++11

LazyData true

Imports Rcpp (>= 0.12.18), rlang, grDevices, graphics

LinkingTo Rcpp

RoxygenNote 7.1.0

URL https://ambient.data-imaginist.com,
https://github.com/thomasp85/ambient

BugReports https://github.com/thomasp85/ambient/issues

NeedsCompilation yes

Author Thomas Lin Pedersen [cre, aut]
(<https://orcid.org/0000-0002-5147-4711>),
Jordan Peck [aut] (Developer of FastNoise)

Repository CRAN

Date/Publication 2020-03-21 17:50:12 UTC

R topics documented:

`ambient-package` *ambient: A Generator of Multidimensional Noise*

Description

Generation of natural looking noise has many application within simulation, procedural generation, and art, to name a few. The 'ambient' package provides an interface to the 'FastNoise' C++ library and allows for efficient generation of perlin, simplex, worley, cubic, value, and white noise with optional pertubation in either 2, 3, or 4 (in case of simplex and white noise) dimensions.

Author(s)

Maintainer: Thomas Lin Pedersen <thomasp85@gmail.com> (ORCID)

Authors:

- Jordan Peck (Developer of FastNoise)

References

https://github.com/Auburns/FastNoise

See Also

Useful links:

- https://ambient.data-imaginist.com
- https://github.com/thomasp85/ambient
- Report bugs at https://github.com/thomasp85/ambient/issues

`billow` *Billow (cloud-like, lumpy) fractal*

Description

The billow fractal is a slight modification of the `fbm()` fractal. Before adding the new layer onto the last, the new layer is modified by taking the absolute value, multiplying by 2, and subtracting one. The result is that the new value will not contain negative values and so will always add on top of the old values. This function is intended to be used in conjunction with `fracture()`

Usage

```
billow(base, new, strength, ...)
```

Arguments

`base`	The prior values to modify
`new`	The new values to modify `base` with
`strength`	A value to modify `new` with before applying it to `base`
`...`	ignored

See Also

Other Fractal functions: `clamped()`, `fbm()`, `ridged()`

Examples

```
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))

grid$simplex <- fracture(gen_simplex, billow, octaves = 8, x = grid$x,
                         y = grid$y)
plot(grid, simplex)
```

clamped *Clamped fractal*

Description

This fractal is a slight variation of `fbm()` fractal. Before adding the new octave to the cumulated values it will clamp it between a minimum and maximum value. This function is intended to be used in conjunction with `fracture()`

Usage

```
clamped(base, new, strength, min = 0, max = Inf, ...)
```

Arguments

`base`	The prior values to modify
`new`	The new values to modify `base` with
`strength`	A value to modify `new` with before applying it to `base`
`min, max`	The upper and lower bounds of the noise values
`...`	ignored

See Also

Other Fractal functions: `billow()`, `fbm()`, `ridged()`

Examples

```
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))

grid$simplex <- fracture(gen_simplex, clamped, octaves = 8, x = grid$x,
                         y = grid$y)
plot(grid, simplex)
```

curl_noise *Generate curl noise*

Description

One of the use cases for fractal noise is to simulate natural phenomena. perlin/simplex noise are e.g. often used to create flow fields, but this can be problematic as they are not divergence-free (particles will concentrate at sinks/gutters in the field). An approach to avoid this is to take the curl of a field instead. The curl operator is ensured to produce divergence-free output, when supplied with continuous fields such as those generated by simplex and perlin noise. The end result is a field that is incompressible, thus modelling fluid dynamics quite well.

Usage

```
curl_noise(
  generator,
  x,
  y,
  z = NULL,
  ...,
  seed = NULL,
  delta = NULL,
  mod = NULL
)
```

Arguments

generator	The noise generating function, such as gen_simplex, or fracture()
x, y, z	The coordinates to generate the curl for as unquoted expressions
...	Further arguments to generator
seed	A seed for the generator. For 2D curl the seed is a single integer and for 3D curl it must be a vector of 3 integers. If NULL the seeds will be random.
delta	The offset to use for the partial derivative of the generator. If NULL, it will be set as 1e-4 of the largest range of the dimensions.
mod	A modification function taking the coordinates along with the output of the generator call and allow modifications of it prior to calculating the curl. The function will get the coordinates as well as a value holding the generator output for each coordinate. If the curl is requested in 2D the value will be a numeric vector and mod() should return a numeric vector of the same length. IF the curl is requested in 3D the value is a list of three numeric vectors (x, y, and z) and mod() should return a list of three vectors of the same length. Passing NULL will use the generator values unmodified.

References

Bridson, Robert. Hourihan, Jim. Nordenstam, Marcus (2007). *Curl-noise for procedural fluid flow*. ACM Transactions on Graphics 26(3): 46. doi:10.1145/1275808.1276435.

See Also

Other derived values: gradient_noise()

Examples

```
grid <- long_grid(seq(0, 1, l = 100), seq(0, 1, l = 100))

# Use one of the generators
grid$curl <- curl_noise(gen_simplex, x = grid$x, y = grid$y)
plot(grid$x, grid$y, type = 'n')
segments(grid$x, grid$y, grid$x + grid$curl$x / 100, grid$y + grid$curl$y / 100)
```

```
# If the curl of fractal noise is needed, pass in `fracture` instead
grid$curl <- curl_noise(fracture, x = grid$x, y = grid$y, noise = gen_simplex,
                        fractal = fbm, octaves = 4)
plot(grid$x, grid$y, type = 'n')
segments(grid$x, grid$y, grid$x + grid$curl$x / 500, grid$y + grid$curl$y / 500)
```

---

fbm *Fractional Brownian Motion fractal*

Description

This is the archetypal fractal used when generating perlin noise. It works simply by adding successive values together to create a final value. As the succesive values are often calculated at increasing frequencies and the strength is often decreasing, it will create the impression of ever-smaller details as you zoom in. This function is intended to be used in conjunction with `fracture()`

Usage

```
fbm(base, new, strength, ...)
```

Arguments

base	The prior values to modify
new	The new values to modify base with
strength	A value to modify new with before applying it to base
...	ignored

See Also

Other Fractal functions: `billow()`, `clamped()`, `ridged()`

Examples

```
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))

grid$simplex <- fracture(gen_simplex, fbm, octaves = 8, x = grid$x, y = grid$y)
plot(grid, simplex)
```

`fracture` *Create fractals of a noise or pattern*

Description

This function allows you to create fractals of a given noise or pattern generator by calculating it repeatedly at changing frequency and combining the results based on a fractal function.

Usage

```
fracture(
  noise,
  fractal,
  octaves,
  gain = ~./2,
  frequency = ~. * 2,
  seed = NULL,
  ...,
  fractal_args = list(),
  gain_init = 1,
  freq_init = 1
)
```

Arguments

`noise`	The noise function to create a fractal from. Must have a `frequency` argument.
`fractal`	The fractal function to combine the generated values with. Can be one of the provided ones or a self-made function. If created by hand it must have the following arguments: • `base`: The current noise values • `new`: The new noise values to combine with `base` • `strength`: The value from `gain` corresponding to the index of `new` • `octave`: The index of `new` And must return a numeric vector of the same length as `new`
`octaves`	The number of generated values to combine
`gain`	The intensity of the generated values at each octave. The interpretation of this is up to the fractal function. Usually the intensity will gradually fall as the frequency increases. Can either be a vector of values or a (lambda) function that returns a new value based on the prior, e.g. `~ . / 2`. The default is often a good starting point though e.g. `ridged()` fractal has been designed with a special gain function.
`frequency`	The frequency to use at each octave. Can either be a vector of values or a function that returns a new value based on the prior. See `gain`.
`seed`	A seed for the noise generator. Will be expanded to the number of octaves so each gets a unique seed.

`...`	arguments to pass on to `generator`
`fractal_args`	Additional arguments to `fractal` as a named list
`gain_init, freq_init`	The gain and frequency for the first octave if `gain` and/or `frequency` are given as a function.

See Also

ambient comes with a range of build in fractal functions: `fbm()`, `billow()`, `ridged()`, `clamped()`

Examples

```
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))

# When noise is generated by it's own it doesn't have fractal properties
grid$clean_perlin <- gen_perlin(grid$x, grid$y)
plot(grid, clean_perlin)

# Use fracture to apply a fractal algorithm to the noise
grid$fractal_perlin <- fracture(gen_perlin, fbm, octaves = 8,
                                x = grid$x, y = grid$y)
plot(grid, fractal_perlin)
```

---

gen_checkerboard *Generate a checkerboard pattern*

Description

This generator supplies 0 or 1 value depending on the provided coordinates position on a checkerboard. The `frequency` determines the number of squares per unit.

Usage

```
gen_checkerboard(x, y = NULL, z = NULL, t = NULL, frequency = 1, ...)
```

Arguments

`x, y, z, t`	The coordinates to get pattern from
`frequency`	The frequency of the generator
`...`	ignored

Value

A numeric vector

See Also

Other Pattern generators: `gen_spheres()`, `gen_waves()`

Examples

```
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))
grid$chess <- gen_checkerboard(grid$x, grid$y)

plot(grid, chess)
```

`gen_spheres` *Generate a pattern of concentric spheres*

Description

This generator creates a pattern of concentric circles centered at 0. Depending on how many dimensions you supply it can be used to generate cylinders and circles as well. The output value is the shortest distance to the nearest sphere normalised to be between -1 and 1. The frequency determines the radius multiplier for each unit sphere.

Usage

```
gen_spheres(x, y = NULL, z = NULL, t = NULL, frequency = 1, ...)
```

Arguments

`x, y, z, t`	The coordinates to get pattern from
`frequency`	The frequency of the generator
`...`	ignored

Value

A numeric vector

See Also

Other Pattern generators: `gen_checkerboard()`, `gen_waves()`

Examples

```
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))
grid$circles <- gen_spheres(grid$x, grid$y)
grid$cylinders <- gen_spheres(grid$x)

plot(grid, circles)
plot(grid, cylinders)
```

gen_waves *Generate a wave pattern*

Description

This generator generates multidimensional waves based on cos to the distance to the center. This means that you can create ripple waves or parallel waves depending on how many dimensions you provide. The output is scaled between -1 and 1 and the frequency determines the number of waves per unit. The result is much like `gen_spheres()` but has smooth transitions at each extreme.

Usage

```
gen_waves(x, y = NULL, z = NULL, t = NULL, frequency = 1, ...)
```

Arguments

`x, y, z, t`	The coordinates to get pattern from
`frequency`	The frequency of the generator
`...`	ignored

Value

A numeric vector

See Also

Other Pattern generators: `gen_checkerboard()`, `gen_spheres()`

Examples

```
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))
grid$ripple <- gen_waves(grid$x, grid$y)
grid$wave <- gen_waves(grid$x)

plot(grid, ripple)
plot(grid, wave)
```

gradient_noise *Calculate the gradient of a scalar field*

Description

The gradient of a scalar field such as those generated by the different noise algorithms in ambient is a vector field encoding the direction to move to get the strongest increase in value. The vectors generated have the properties of being perpendicular on the contour line drawn through that point. Take note that the returned vector field flows upwards, i.e. points toward the steepest ascend, rather than what is normally expected in a gravitational governed world.

Usage

```
gradient_noise(
  generator,
  x,
  y,
  z = NULL,
  t = NULL,
  ...,
  seed = NULL,
  delta = NULL
)
```

Arguments

generator	The noise generating function, such as gen_simplex, or fracture()
x, y, z, t	The coordinates to generate the gradient for as unquoted expressions
...	Further arguments to generator
seed	A seed for the generator.
delta	The offset to use for the partial derivative of the generator. If NULL, it will be set as 1e-4 of the largest range of the dimensions.

See Also

Other derived values: curl_noise()

Examples

```
grid <- long_grid(seq(0, 1, l = 100), seq(0, 1, l = 100))

# Use one of the generators
grid$gradient <- gradient_noise(gen_simplex, x = grid$x, y = grid$y)
plot(grid$x, grid$y, type = 'n')
segments(grid$x, grid$y, grid$x + grid$gradient$x / 100, grid$y + grid$gradient$y / 100)
```

long_grid *Create a long format grid*

Description

This function creates a 1-4 dimensional grid in long format, with the cell positions encoded in the x, y, z, and t columns. A long_cell object is the base class for the tidy interface to ambient, and allows a very flexible approach to pattern generation at the expense of slightly lower performance than the noise_* functions that maps directly to the underlying C++ code.

Usage

```
long_grid(x, y = NULL, z = NULL, t = NULL)

grid_cell(grid, dim, ...)

## S3 method for class 'long_grid'
as.array(x, value, ...)

## S3 method for class 'long_grid'
as.matrix(x, value, ...)

## S3 method for class 'long_grid'
as.raster(x, value, ...)

slice_at(grid, ...)
```

Arguments

x, y, z, t	For long_grid() vectors of grid cell positions for each dimension. The final dimensionality of the object is determined by how many arguments are given. For slice_at() an integer defining the index at the given dimension to extract.
grid	A long_grid object
dim	The dimension to get the cell index at, either as an integer or string.
...	Arguments passed on to methods (ignored)
value	The unquoted value to use for filling out the array/matrix

Examples

```
grid <- long_grid(1:10, seq(0, 1, length = 6), c(3, 6))

# Get which row each cell belongs to
grid_cell(grid, 2) # equivalent to grid_cell(grid, 'y')

# Convert the long_grid to an array and fill with the x position
as.array(grid, x)
```

```
# Extract the first column
slice_at(grid, x = 1)

# Convert the first column to a matrix filled with y position
as.matrix(slice_at(grid, x = 1), y)
```

`modifications` *Simply value modifications*

Description

Most modifications of values in a long_grid are quite simple due to the wealth of vectorised functions avaliable in R. ambient provides a little selection of handy functions to compliment these

Usage

```
blend(x, y, mask)

normalise(x, from = range(x), to = c(0, 1))

normalize(x, from = range(x), to = c(0, 1))

cap(x, lower = 0, upper = 1)
```

Arguments

`x, y`	Aalues to modify
`mask`	A vector of the same length as `x` and `y`. Assumed to be between 0 and 1 (values outside of this range is capped). The closer to 1 the more of `x` will be used and the closer to 0 the more of `y` will be used
`from`	The range of `x` to use for normalisation
`to`	The output domain to normalise to
`lower, upper`	The lower and upper bounds to cap to

Examples

```
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))
grid$chess <- gen_checkerboard(grid$x, grid$y)
grid$noise <- gen_perlin(grid$x, grid$y)
grid$ripple <- gen_waves(grid$x, grid$y)

# Blend two values based on a third
grid$mix <- blend(grid$noise, grid$ripple, grid$chess)
plot(grid, mix)
```

```
# Cap values between 0 and 1
plot(grid, cap(noise))
```

`noise_cubic` *Cubic noise generator*

Description

Cubic noise is a pretty simple alternative to perlin and simplex noise. In essence it takes a low resolution white noise and scales it up using cubic interpolation. This approach means that while cubic noise is smooth, it is much more random than perlin and simplex noise.

Usage

```
noise_cubic(
  dim,
  frequency = 0.01,
  fractal = "fbm",
  octaves = 3,
  lacunarity = 2,
  gain = 0.5,
  pertubation = "none",
  pertubation_amplitude = 1
)

gen_cubic(x, y = NULL, z = NULL, frequency = 1, seed = NULL, ...)
```

Arguments

`dim`	The dimensions (height, width, (and depth)) of the noise to be generated. The length determines the dimensionality of the noise.
`frequency`	Determines the granularity of the features in the noise.
`fractal`	The fractal type to use. Either `'none'`, `'fbm'` (default), `'billow'`, or `'rigid-multi'`. It is suggested that you experiment with the different types to get a feel for how they behaves.
`octaves`	The number of noise layers used to create the fractal noise. Ignored if `fractal = 'none'`. Defaults to `3`.
`lacunarity`	The frequency multiplier between successive noise layers when building fractal noise. Ignored if `fractal = 'none'`. Defaults to `2`.
`gain`	The relative strength between successive noise layers when building fractal noise. Ignored if `fractal = 'none'`. Defaults to `0.5`.
`pertubation`	The pertubation to use. Either `'none'` (default), `'normal'`, or `'fractal'`. Defines the displacement (warping) of the noise, with `'normal'` giving a smooth warping and `'fractal'` giving a more eratic warping.

pertubation_amplitude
: The maximal pertubation distance from the origin. Ignored if `pertubation = 'none'`. Defaults to 1.

`x, y, z`
: Coordinates to get noise value from

`seed`
: The seed to use for the noise. If `NULL` a random seed will be used

`...`
: ignored

Value

For `noise_cubic()` a matrix if `length(dim) == 2` or an array if `length(dim) == 3`. For `gen_cubic()` a numeric vector matching the length of the input.

Examples

```
# Basic use
noise <- noise_cubic(c(100, 100))

plot(as.raster(normalise(noise)))

# Using the generator
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))
grid$noise <- gen_cubic(grid$x, grid$y)
plot(grid, noise)
```

---

noise_perlin *Perlin noise generator*

Description

This function generates either 2 or 3 dimensional perlin noise, with optional pertubation and fractality. Perlin noise is one of the most well known gradient noise algorithms and have been used extensively as the basis for generating landscapes and textures, as well as within generative art. The algorithm was developed by Ken Perlin in 1983.

Usage

```
noise_perlin(
  dim,
  frequency = 0.01,
  interpolator = "quintic",
  fractal = "fbm",
  octaves = 3,
  lacunarity = 2,
  gain = 0.5,
  pertubation = "none",
  pertubation_amplitude = 1
```

```
)

gen_perlin(
  x,
  y = NULL,
  z = NULL,
  frequency = 1,
  seed = NULL,
  interpolator = "quintic",
  ...
)
```

Arguments

`dim`	The dimensions (height, width, (and depth)) of the noise to be generated. The length determines the dimensionality of the noise.
`frequency`	Determines the granularity of the features in the noise.
`interpolator`	How should values between sampled points be calculated? Either `'linear'`, `'hermite'`, or `'quintic'` (default), ranging from lowest to highest quality.
`fractal`	The fractal type to use. Either `'none'`, `'fbm'` (default), `'billow'`, or `'rigid-multi'`. It is suggested that you experiment with the different types to get a feel for how they behaves.
`octaves`	The number of noise layers used to create the fractal noise. Ignored if `fractal = 'none'`. Defaults to `3`.
`lacunarity`	The frequency multiplier between successive noise layers when building fractal noise. Ignored if `fractal = 'none'`. Defaults to `2`.
`gain`	The relative strength between successive noise layers when building fractal noise. Ignored if `fractal = 'none'`. Defaults to `0.5`.
`pertubation`	The pertubation to use. Either `'none'` (default), `'normal'`, or `'fractal'`. Defines the displacement (warping) of the noise, with `'normal'` giving a smooth warping and `'fractal'` giving a more eratic warping.
`pertubation_amplitude`	The maximal pertubation distance from the origin. Ignored if `pertubation = 'none'`. Defaults to `1`.
`x, y, z`	Coordinates to get noise value from
`seed`	The seed to use for the noise. If `NULL` a random seed will be used
`...`	ignored

Value

For `noise_perlin()` a matrix if `length(dim) == 2` or an array if `length(dim) == 3`. For `gen_perlin()` a numeric vector matching the length of the input.

References

Perlin, Ken (1985). *An Image Synthesizer.* SIGGRAPH Comput. Graph. 19 (0097-8930): 287–296. doi:10.1145/325165.325247.

Examples

```
# Basic use
noise <- noise_perlin(c(100, 100))

plot(as.raster(normalise(noise)))

# Using the generator
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))
grid$noise <- gen_perlin(grid$x, grid$y)
plot(grid, noise)
```

`noise_simplex` *Simplex noise generator*

Description

Simplex noise has been developed by Ken Perlin, the inventor of perlin noise, in order to address some of the shortcomings he saw in perlin noise. Compared to perlin noise, simplex noise has lower computational complexity, making it feasable for dimensions above 3 and has no directional artifacts.

Usage

```
noise_simplex(
  dim,
  frequency = 0.01,
  interpolator = "quintic",
  fractal = "fbm",
  octaves = 3,
  lacunarity = 2,
  gain = 0.5,
  pertubation = "none",
  pertubation_amplitude = 1
)

gen_simplex(x, y = NULL, z = NULL, t = NULL, frequency = 1, seed = NULL, ...)
```

Arguments

`dim`	The dimensions (height, width, (and depth, (and time))) of the noise to be generated. The length determines the dimensionality of the noise.
`frequency`	Determines the granularity of the features in the noise.
`interpolator`	How should values between sampled points be calculated? Either `'linear'`, `'hermite'`, or `'quintic'` (default), ranging from lowest to highest quality.

`fractal`	The fractal type to use. Either `'none'`, `'fbm'` (default), `'billow'`, or `'rigid-multi'`. It is suggested that you experiment with the different types to get a feel for how they behaves.
`octaves`	The number of noise layers used to create the fractal noise. Ignored if `fractal = 'none'`. Defaults to `3`.
`lacunarity`	The frequency multiplier between successive noise layers when building fractal noise. Ignored if `fractal = 'none'`. Defaults to `2`.
`gain`	The relative strength between successive noise layers when building fractal noise. Ignored if `fractal = 'none'`. Defaults to `0.5`.
`pertubation`	The pertubation to use. Either `'none'` (default), `'normal'`, or `'fractal'`. Defines the displacement (warping) of the noise, with `'normal'` giving a smooth warping and `'fractal'` giving a more eratic warping.
`pertubation_amplitude`	The maximal pertubation distance from the origin. Ignored if `pertubation = 'none'`. Defaults to `1`.
`x, y, z, t`	Coordinates to get noise value from
`seed`	The seed to use for the noise. If `NULL` a random seed will be used
`...`	ignored

Value

For `noise_simplex()` a matrix if `length(dim) == 2` or an array if `length(dim) >= 3`. For `gen_simplex()` a numeric vector matching the length of the input.

References

Ken Perlin, (2001) *Noise hardware*. In Real-Time Shading SIGGRAPH Course Notes, Olano M., (Ed.)

Examples

```
# Basic use
noise <- noise_simplex(c(100, 100))

plot(as.raster(normalise(noise)))

# Using the generator
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))
grid$noise <- gen_simplex(grid$x, grid$y)
plot(grid, noise)
```

`noise_value` *Value noise generator*

Description

Value noise is a simpler version of cubic noise that uses linear interpolation between neighboring grid points. This creates a more distinct smooth checkerboard pattern than cubic noise, where interpolation takes all the surrounding grid points into accout.

Usage

```
noise_value(
  dim,
  frequency = 0.01,
  interpolator = "quintic",
  fractal = "fbm",
  octaves = 3,
  lacunarity = 2,
  gain = 0.5,
  pertubation = "none",
  pertubation_amplitude = 1
)

gen_value(
  x,
  y = NULL,
  z = NULL,
  frequency = 1,
  seed = NULL,
  interpolator = "quintic",
  ...
)
```

Arguments

`dim`	The dimensions (height, width, (and depth)) of the noise to be generated. The length determines the dimensionality of the noise.
`frequency`	Determines the granularity of the features in the noise.
`interpolator`	How should values between sampled points be calculated? Either `'linear'`, `'hermite'`, or `'quintic'` (default), ranging from lowest to highest quality.
`fractal`	The fractal type to use. Either `'none'`, `'fbm'` (default), `'billow'`, or `'rigid-multi'`. It is suggested that you experiment with the different types to get a feel for how they behaves.
`octaves`	The number of noise layers used to create the fractal noise. Ignored if `fractal = 'none'`. Defaults to `3`.

lacunarity	The frequency multiplier between successive noise layers when building fractal noise. Ignored if `fractal = 'none'`. Defaults to `2`.
gain	The relative strength between successive noise layers when building fractal noise. Ignored if `fractal = 'none'`. Defaults to `0.5`.
pertubation	The pertubation to use. Either `'none'` (default), `'normal'`, or `'fractal'`. Defines the displacement (warping) of the noise, with `'normal'` giving a smooth warping and `'fractal'` giving a more eratic warping.
pertubation_amplitude	The maximal pertubation distance from the origin. Ignored if `pertubation = 'none'`. Defaults to `1`.
x, y, z	Coordinates to get noise value from
seed	The seed to use for the noise. If `NULL` a random seed will be used
...	ignored

Value

For `noise_value()` a matrix if `length(dim) == 2` or an array if `length(dim) == 3`. For `gen_value()` a numeric vector matching the length of the input.

Examples

```
# Basic use
noise <- noise_value(c(100, 100))

plot(as.raster(normalise(noise)))

# Using the generator
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))
grid$noise <- gen_value(grid$x, grid$y)
plot(grid, noise)
```

---

noise_white *White noise generator*

Description

White noise is a random noise with equal intensities at different frequencies. It is most well-known as what appeared on old televisions when no signal was found.

Usage

```
noise_white(
  dim,
  frequency = 0.01,
  pertubation = "none",
```

```
  pertubation_amplitude = 1
)

gen_white(x, y = NULL, z = NULL, t = NULL, frequency = 1, seed = NULL, ...)
```

Arguments

dim	The dimensions (height, width, (and depth, (and time))) of the noise to be generated. The length determines the dimensionality of the noise.
frequency	Determines the granularity of the features in the noise.
pertubation	The pertubation to use. Either 'none' (default), 'normal', or 'fractal'. Defines the displacement (warping) of the noise, with 'normal' giving a smooth warping and 'fractal' giving a more eratic warping.
pertubation_amplitude	The maximal pertubation distance from the origin. Ignored if pertubation = 'none'. Defaults to 1.
x, y, z, t	Coordinates to get noise value from
seed	The seed to use for the noise. If NULL a random seed will be used
...	ignored

Value

For noise_white() a matrix if length(dim) == 2 or an array if length(dim) >= 3. For gen_white() a numeric vector matching the length of the input.

Examples

```
# Basic use
noise <- noise_white(c(100, 100))

plot(as.raster(normalise(noise)))

# Using the generator
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))
grid$noise <- gen_white(grid$x, grid$y)
plot(grid, noise)
```

---

noise_worley *Worley (cell) noise generator*

Description

Worley noise, sometimes called cell (or cellular) noise, is quite distinct due to it's kinship to voronoi tesselation. It is created by sampling random points in space and then for any point in space measure the distance to the closest point. The noise can be modified further by changing either the distance measure or by combining multiple distances. The noise algorithm was developed by Steven Worley in 1996 and has been used to simulated water and stone textures among other things.

Usage

```
noise_worley(
  dim,
  frequency = 0.01,
  distance = "euclidean",
  fractal = "none",
  octaves = 3,
  lacunarity = 2,
  gain = 0.5,
  value = "cell",
  distance_ind = c(1, 2),
  jitter = 0.45,
  pertubation = "none",
  pertubation_amplitude = 1
)

gen_worley(
  x,
  y = NULL,
  z = NULL,
  frequency = 1,
  seed = NULL,
  distance = "euclidean",
  value = "cell",
  distance_ind = c(1, 2),
  jitter = 0.45,
  ...
)
```

Arguments

`dim`	The dimensions (height, width, (and depth)) of the noise to be generated. The length determines the dimensionality of the noise.
`frequency`	Determines the granularity of the features in the noise.
`distance`	The distance measure to use, either `'euclidean'` (default), `'manhattan'`, or `'natural'` (a mix of the two)
`fractal`	The fractal type to use. Either `'none'`, `'fbm'` (default), `'billow'`, or `'rigid-multi'`. It is suggested that you experiment with the different types to get a feel for how they behaves.
`octaves`	The number of noise layers used to create the fractal noise. Ignored if `fractal = 'none'`. Defaults to `3`.
`lacunarity`	The frequency multiplier between successive noise layers when building fractal noise. Ignored if `fractal = 'none'`. Defaults to `2`.
`gain`	The relative strength between successive noise layers when building fractal noise. Ignored if `fractal = 'none'`. Defaults to `0.5`.
`value`	The noise value to return. Either

- 'value' (default) A random value associated with the closest point
- 'distance' The distance to the closest point
- 'distance2' The distance to the nth closest point (n given by distance_ind[1])
- 'distance2add' Addition of the distance to the nth and mth closest point given in distance_ind
- 'distance2sub' Substraction of the distance to the nth and mth closest point given in distance_ind
- 'distance2mul' Multiplication of the distance to the nth and mth closest point given in distance_ind
- 'distance2div' Division of the distance to the nth and mth closest point given in distance_ind

distance_ind — Reference to the nth and mth closest points that should be used when calculating value.

jitter — The maximum distance a point can move from its start position during sampling of cell points.

pertubation — The pertubation to use. Either 'none' (default), 'normal', or 'fractal'. Defines the displacement (warping) of the noise, with 'normal' giving a smooth warping and 'fractal' giving a more eratic warping.

pertubation_amplitude — The maximal pertubation distance from the origin. Ignored if pertubation = 'none'. Defaults to 1.

x, y, z — Coordinates to get noise value from

seed — The seed to use for the noise. If NULL a random seed will be used

... — ignored

Value

For noise_worley() a matrix if length(dim) == 2 or an array if length(dim) == 3. For gen_worley() a numeric vector matching the length of the input.

References

Worley, Steven (1996). *A cellular texture basis function*. Proceedings of the 23rd annual conference on computer graphics and interactive techniques. pp. 291–294. ISBN 0-89791-746-4

Examples

```
# Basic use
noise <- noise_worley(c(100, 100))

plot(as.raster(normalise(noise)))

# Using the generator and another value metric
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))
grid$noise <- gen_worley(grid$x, grid$y, value = 'distance')
plot(grid, noise)
```

`ridged` *Ridged-Multi fractal*

Description

This fractal is slightly more complex than the regular `fbm()` fractal. It uses the prior octave to modify the values of the current octave before adding it to the cumulating values. The result of this is that the final values will show steep hills and larger smooth areas, resembling mountain ranges. This function is intended to be used in conjunction with `fracture()`

Usage

```
ridged(base, new, strength, octave, offset = 1, gain = 2, ...)

spectral_gain(h = 1, lacunarity = 2)
```

Arguments

`base`	The prior values to modify
`new`	The new values to modify `base` with
`strength`	A value to modify `new` with before applying it to `base`
`octave`	The current octave
`offset`	The new values are first modified by `(offset -abs(new))^2`
`gain`	A value to multiply the old octave by before using it to modify the new octave
`...`	ignored
`h`	Each successive gain is raised to the power of `-h`
`lacunarity`	A multiplier to apply to the previous value before raising it to the power of `-h`

Details

The ridged fractal was designed with a slightly more complex gain sequence in mind, and while any sequence or generator would work `fracture()` should be called with `gain = spectral_gain()` to mimick the original intention of the fractal.

See Also

Other Fractal functions: `billow()`, `clamped()`, `fbm()`

Examples

```
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))

grid$simplex <- fracture(gen_simplex, ridged, octaves = 8,
                         gain = spectral_gain(), x = grid$x, y = grid$y)
plot(grid, simplex)
```

`trans_affine` *Apply linear transformation to a long_grid*

Description

This function allows you to calculate linear transformations of coordinates in a long_grid object. You can either pass in a transformation matrix or a trans object as produced by `ggforce::linear_trans(...)`. The latter makes it easy to stack multiple transformations into one, but require the ggforce package.

Usage

```
trans_affine(x, y, ...)

rotate(angle = 0)

stretch(x0 = 0, y0 = 0)

shear(x0 = 0, y0 = 0)

translate(x0 = 0, y0 = 0)

reflect(x0 = 0, y0 = 0)
```

Arguments

`x, y`	The coordinates to transform
`...`	A sequence of transformations
`angle`	An angle in radians
`x0`	the transformation magnitude in the x-direction
`y0`	the transformation magnitude in the x-direction

Linear Transformations

The following transformation matrix constructors are supplied, but you can also provide your own 3x3 matrices to `translate()`

- `rotate()`: Rotate coordinates by `angle` (in radians) around the center counter-clockwise.
- `stretch()`: Stretches the x and/or y dimension by multiplying it with `x0`/`y0`.
- `shear()`: Shears the x and/or y dimension by `x0`/`y0`.
- `translate()`: Moves coordinates by `x0`/`y0`.
- `reflect()`: Reflects coordinates through the line that goes through 0, 0 and x0, y0.

Examples

```
grid <- long_grid(seq(1, 10, length.out = 1000), seq(1, 10, length.out = 1000))
grid$trans <- trans_affine(grid$x, grid$y, rotate(pi/3), shear(-2), rotate(-pi/3))
grid$chess <- gen_checkerboard(grid$trans$x, grid$trans$y)

plot(grid, chess)
```

Index